1.10 — Introduction to expressions

ALEX[1] MARCH 11, 2024

Expressions

Consider the following series of statements:

```
// five() is a function that returns the value 5
int five()
{
    return 5;
}

int main()
{
    int a{ 2 };             // initialize variable a with literal value 2
    int b{ 2 + 3 };         // initialize variable b with computed value 5
    int c{ (2 * 3) + 4 };   // initialize variable c with computed value 10
    int d{ b };             // initialize variable d with variable value 5
    int e{ five() };        // initialize variable e with function return
value 5

    return 0;
}
```

Each of these statements defines a new variable and initializes it with a value. Note that the initializers shown above make use of a variety of different constructs: literals, variables, operators, and function calls. Somehow, C++ is converting all of these different things into a single value that can then be used as the initialization value for the variable.

What do all of these initializers have in common? They make use of an expression.

An **expression** is a sequence of literals, variables, operators, and function calls that calculates a single value. The process of executing an expression is called **evaluation**, and the single value produced is called the **result** of the expression.

Related content

While most expressions are used to calculate a value, expressions can also identify an object (which can be evaluated to get the value held by the object) or a function (which can be called to get the value returned by the function). We talk more about this in lesson 12.2 -- Value categories (lvalues and rvalues) (https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/)[2].

For now, we'll assume all expressions calculate values.

When an expression is evaluated, each of the terms inside the expression are evaluated, until a single value remains. Here are some examples of different kinds of expressions, with comments indicating how they evaluate:

```
2               // 2 is a literal that evaluates to value 2
"Hello world!"  // "Hello world!" is a literal that evaluates to text "Hello world!"
x               // x is a variable that evaluates to the value of x
2 + 3           // operator+ uses operands 2 and 3 to evaluate to value 5
five()          // evaluates to the return value of function five()
```

As you can see, literals evaluate to their own values. Variables evaluate to the value of the variable. Operators (such as `operator+`) use their operands to evaluate to some other value. We haven't covered function calls yet, but in the context of an expression, function calls evaluate to whatever value the function returns.

Expressions involving operators with side effects are a little more tricky:

```
x = 5           // x = 5 has side effect of assigning 5 to x, evaluates to x
x = 2 + 3       // has side effect of assigning 5 to x, evaluates to x
std::cout << x  // has side effect of printing value of x to console, evaluates to std::cout
```

Note that expressions do not end in a semicolon, and cannot be compiled by themselves. For example, if you were to try compiling the expression `x = 5`, your compiler would complain (probably about a missing semicolon). Rather, expressions are always evaluated as part of statements.

For example, take this statement:

```
int x{ 2 + 3 }; // 2 + 3 is an expression that has no semicolon -- the semicolon is at the end of the statement containing the expression
```

If you were to break this statement down into its syntax, it would look like this:

`type identifier { expression };`

type could be any valid type (we chose `int`). *identifier* could be any valid name (we chose `x`). And *expression* could be any valid expression (we chose `2 + 3`, which uses two literals and an operator).

Key insight

Wherever you can use a single value in C++, you can use a value-producing expression instead, and the expression will be evaluated to produce a single value.

Expression statements

Certain expressions (like `x = 5`) are useful for their side effects (in this case, to assign the value `5` to the variable `x`). However, we mentioned above that expressions cannot be executed by themselves -- they must exist as part of a statement. So how can we use such expressions?

Fortunately, it's easy to convert any expression into an equivalent statement. An **expression statement** is a statement that consists of an expression followed by a semicolon. When the expression statement is executed, the expression will be evaluated.

Thus, we can take any expression (such as `x = 5`), and turn it into an expression statement (`x = 5;`) that will compile.

When an expression is used in an expression statement, any return value generated by the expression is discarded (because it is not used).

Useless expression statements

We can also make expression statements that compile but have no effect. For example, the expression statement (`2 * 3;`) is an expression statement whose expression evaluates to the result value of *6*, which is then discarded. While syntactically valid, such expression statements are useless. Some compilers (such as gcc and Clang) will produce warnings if they can detect that an expression statement is useless.

Subexpressions, full expressions, and compound expressions

We occasionally need to talk about specific kinds of expressions. For this purpose, we will define some related terms.

Consider the following expressions:

```
2               // 2 is a literal that evaluates to value 2
2 + 3           // 2 + 3 uses operator + to evaluate to value 5
x = 4 + 5       // 4 + 5 evaluates to value 9, which is then assigned to variable x
```

Simplifying a bit, a **subexpression** is an expression used as an operand. For example, the subexpressions of `x = 4 + 5` are `x` and `4 + 5`. The subexpressions of `4 + 5` are `4` and `5`.

A **full expression** is an expression that is not a subexpression. All three expressions above (`2`, `2 + 3`, and `x = 4 + 5`) are full expressions.

In casual language, a **compound expression** is an expression that contains two or more uses of operators. `x = 4 + 5` is a compound expression because it contains two uses of operators (`operator=` and `operator+`). `2` and `2 + 3` are not compound expressions.

Quiz time

Question #1

What is the difference between a statement and an expression?

Hide Solution (javascript:void(0))[3]

> Statements are used when we want the program to perform an action. Expressions are used when we want the program to calculate a value.

Question #2

Indicate whether each of the following lines are *statements that do not contain expressions*, *statements that contain expressions*, or are *expression statements*.

a)

```
1 | int x;
```

Hide Solution (javascript:void(0))[3]

> Statement does not contain an expression (this is just a variable definition).

b)

```
1 | int x = 5;
```

Hide Solution (javascript:void(0))[3]

> Statement contains an expression. `int x` is a variable definition. The `=` is part of the syntax for copy initialization. The initializer to the right of the equals sign is an expression.

c)

```
1 | x = 5;
```

Hide Solution (javascript:void(0))[3]

> Expression statement. `x = 5` is a call to `operator=` with two operands: `x` and `5`. The semicolon makes it an expression statement.

d) Extra credit:

```
1 | foo(); // foo is a function
```

Hide Solution (javascript:void(0))[3]

> Function calls are part of an expression, so this is an expression statement.

e) Extra credit:

```
std::cout << x; // Hint: operator<< is a binary operator.
```

Hide Solution (javascript:void(0))[3]

> `operator<<` is a binary operator, so `std::cout` must be the left-hand operand, and `x` must be the right-hand operand. Since that's the entire statement, this must be an expression statement.

Question #3

Determine what values the following program outputs. Do not compile this program. Just work through it line by line in your head.

```
#include <iostream>

int main()
{
    std::cout << 2 + 3 << '\n';

    int x{ 6 };
    int y{ x - 2 };
    std::cout << y << '\n';

    int z{ };
    z = x;
    std::cout << z * x << '\n';

    return 0;
}
```

Hide Solution (javascript:void(0))[3]

> 5
> 4
> 36

Next lesson
1.11 Developing your first program

4

Back to table of contents

5

Previous lesson
1.9 Introduction to literals and operators

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

280 COMMENTS Newest

Ryan Rawlings
February 24, 2024 9:48 pm

On Question 3, If a new programmer misinterpreted line 8 as updating the value of `x`, then they'd still get the correct answer, because `4 - 4` evaluates to zero just as `6 - 6` does.

1 Reply

Alex Author
Reply to Ryan Rawlings[11] February 27, 2024 6:21 pm

Changed subtraction to multiplication. Thanks for pointing this out.

4 Reply

IceFloe
February 18, 2024 5:05 pm

I didn't understand a bit why `std::cout << x;` is an expression if `operator<<` doesn't calculate any values but just prints what's on the right?

0 Reply

Alex Author

Reply to IceFloe [12] February 18, 2024 9:31 pm

An expression doesn't HAVE to calculate a value -- it can also evaluate to an object. `std::cout << x` evaluates to the object `std::cout`.

This allows us to do things like `std::cout << x << y << z` and have it work properly. If `operator<<` returned nothing, then after `x` was output, it wouldn't know where to send `y` and `z`.

4 Reply

Elias

February 4, 2024 5:42 pm

Hello Alex,

First of all, thank you so much for your great lessons!

I have a question..

You say that an expression is a combination of literals, variables, operators, and function calls that calculates a single value - and you follow it up by 5 examples.

```
int main()
{
int a{ 2 }; // initialize variable a with literal value 2
int b{ 2 + 3 }; // initialize variable b with computed value 5
int c{ (2 * 3) + 4 }; // initialize variable c with computed value 10
int d{ b }; // initialize variable d with variable value 5
int e{ five() }; // initialize variable e with function return value 5

return 0;
}
```

How is the first statement int a{ 2 } an expression when it doesn't involve a calculation nor a combination of what you stated. I see it as a declaration and initialization statement, but it's not strictly an expression in the same way that the other statements are. Every other example make sense to me, just not this one.

What am I missing?

Last edited 2 months ago by Elias

0 Reply

Alex Author

Reply to Elias [13] February 5, 2024 4:16 pm

The initializers are expressions (the variable declarations are statements).

I replaced the term "combination" with "sequence", as order matters. `2` is an expression containing a single literal. `b` is an expression containing a single variable. `five()` is an expression containing a single function call. `2 + 3` is an expression containing one operator (and two operands). `(2 * 3) + 4` is a compound expression containing two operators.

2 Reply

Elias

Reply to Alex [14] February 5, 2024 5:17 pm

Thank you for your fast reply. Much appreciated!

0 Reply

Mridul Thakur

February 2, 2024 9:42 pm

If we do that:

```
1 int y;
2 int x {y = 56};
3 y = 34;
4 std::cout << x;
```

y = 56 return y itself. Then, if we change the value of y it should change the value of x too as x is equal to y. Then, why doesn't it change?

0 Reply

Alex Author

Reply to Mridul Thakur [15] February 3, 2024 11:43 am

`int x { y = 56 }` is the equivalent of:

```
1 y = 56;
2 int x { y };
```

When `x` is initialized, it is initialized with the *value* of y. `x` does not become an alias for `y`. If you want that, you need to make `x` a reference, which we cover in lesson https://www.learncpp.com/cpp-tutorial/lvalue-references/

2 Reply

Mridul Thakur

Reply to Alex [16] February 5, 2024 6:54 pm

thanks for your reply.

0 Reply

Luis

January 25, 2024 8:24 am

Are the {} braces an operator in the context of this code?:

```
1 | int x{};
```

The = sign is an operator in the following code and they do the same thing:

```
1 | int x = 0;
```

0 Reply

Alex Author

Reply to Luis [17] January 25, 2024 9:48 pm

No, the {} braces are part of the list initialization syntax, not an operator.
Similarly, the = in an initialization is part of the copy initialization syntax, NOT a use of the = operator.

0 Reply

Mamad

Reply to Alex [18] February 1, 2024 11:14 am

thanks

Last edited 2 months ago by Mamad

0 Reply

Peter

January 24, 2024 11:09 pm

It took me a bit to remind myself that z = x it reads from right to left. So it's not 0 - 0 = 0 but is

4 - 4 with result 0.

Last edited 2 months ago by Peter

0 Reply

tiro
December 24, 2023 7:00 am

Hello ,

I feel like it would be insightful to add that an operation is an expression (if I understood correctly).

You could put it in parenthesis after "An expression is a combination of literals, variables, operators, and function calls that calculates a single value."

I'm slow to understand things and it took me a bit of time to make the link.

0 Reply

Alex Author
Reply to tiro [19] December 26, 2023 7:05 pm

In C++, we typically talk about operators rather than operations, and operators are already in the list of things that expressions can be composed of.

1 Reply

tiro
Reply to Alex [20] December 28, 2023 12:21 am

Thank you for answering .

0 Reply

zestylegume3000
Reply to tiro [21] January 12, 2024 3:03 pm

bro got a response from da big dawg himself

4 Reply

AbeerOrTwo
December 14, 2023 6:44 pm

Got numbah three right

4 Reply

David Pinheiro
November 22, 2023 4:53 am

In the previous lesson we saw: Operand -> Operator -> Return Value and/or Side Effect

One such example was: 2 + 3 produces return value 5

In this lesson we say: 2 + 3 // operator+ uses operands 2 and 3 to evaluate to value 5

So what is the difference between "producing a return value" or "evaluating to something" ?

0 Reply

David Pinheiro
Reply to David Pinheiro [22] November 22, 2023 5:16 am

Is it that operators produce side effects and/or return values and expressions evaluate to something? Yet this lesson says:

"When an expression is used in an expression statement, any return value generated by the expression is discarded"

So expressions not only evaluate to something but also return something, or is this the same?

Also, lesson 1.9 says "Thus, x = 5 returns x" -> should this instead say "returns the value of x"? (since "x" is the variable)

And if then I do "return x" to close the function, why would it return "5" the (value in "x") and not the "x" variable itself?

Last edited 4 months ago by David Pinheiro

0 Reply

Alex Author
Reply to David Pinheiro [23] November 24, 2023 10:59 am

"evaluate to" and "return" mean the same thing in context of the evaluation of expressions.

`x = 5` actually returns `x` (not just the value of `x`) so that you can do things that require an object (not just a value). For example `(x = 5) = 6;` is legal (but silly) because `x = 5` returns `x`, which we can then assign value 6 to. If `x = 5` returned value `5` instead, then the resulting partially evaluated expression would be `5 = 6` which makes no sense.

> We discuss this in more detail in lesson https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/
>
> 1 Reply

Jared

November 17, 2023 8:37 pm

Im confused on what literal means. I thought literals are constants because in your code:

```
int five()
{
return 5;
}

int main()
{
int a{ 2 }; // initialize variable a with literal value 2
int b{ 2 + 3 }; // initialize variable b with computed value 5
int c{ (2 * 3) + 4 }; // initialize variable c with computed value 10
int d{ b }; // initialize variable d with variable value 5
int e{ five() }; // initialize variable e with function return value 5

return 0;
}
```

you said "initialize variable a with literal value 2" which confused me because the literal value 2 can be changed later on in the program if it is assigned to something else, no?

Previously, you gave this example:

```
int main()
{
std::cout << 5 << '\n'; // print the value of a literal

int x { 5 };
std::cout << x << '\n'; // print the value of a variable
return 0;
}
```

which made sense to me because it directly initialized variable x with the value 5. So I guess my question is a constant literal value can be initialized in a variable, but can also still be changed?

0 Reply

> 
>
> **Jared**
>
> Reply to Jared [24] November 17, 2023 8:59 pm

Rephrased question: is

int a{2};

going to print a value of a variable like it did in the example

int x{5};

or will it print the value of a literal?

0 Reply

Jared

Reply to Jared [25] November 19, 2023 2:22 pm

I ended up finding the answer to my question. Thank you for all of this free info!

0 Reply

Alex Author

Reply to Jared [25] November 18, 2023 2:41 pm

Literals are constants in your code. Both `2` and `5` are literals, and `a` and `x` are variables. When we do `int a { 2 }`, variable `a` is initialized with literal value `2`. That means the value `2` is copied into the memory allocated for `a`. We overwrite the value `2` with some other value later if we wish.

1 Reply

Jared

Reply to Alex [26] November 19, 2023 9:24 pm

I didn't see this comment thank you for the clarification and fast reply.

0 Reply

tatadott

October 23, 2023 1:16 pm

This lesson cleared up the differences between expressions and statements for me. Thanks Alex!

0 Reply

James

October 20, 2023 9:03 am

I was taught functions had three components like so.

```
#include <iostream>

int five();

int main() {
// std::cout << "Hello World!\n";
int a{2};
int b{2+3};
int c{(2*3)+4};
int d{b};
int e{five()};
std::cout<<e<<" "<<c;
}

int five(){
return 5;
}
```

is anything wrong with this?

it runs just fine?

0 Reply

Aaron Nahshon Lynch

Reply to James [27] November 8, 2023 9:19 am

int e{five()}; why not do int e = 5; ?

0 Reply

Alex Author

Reply to James [27] October 21, 2023 1:38 pm

Not sure what you mean by "three components".

But yes, this is fine.

0 Reply

Vladimir

October 17, 2023 1:36 pm

Question. First it says.

Here are some examples of different kinds of expressions, with comments indicating how they evaluate:

```
1 | x              // x is a variable that evaluates to the value of x
```

So, x is an expression. Then, in the quiz it says:

```
1 | int x;
```

Statement does not contain an expression (this is just a variable definition).

But isn't x an expression?

Buy the way, huge thanks for your work!

1 Reply

park

Reply to Vladimir [28] October 18, 2023 12:28 am

In the former case, we assume that `x` is a variable that is defined somewhere before the expression - if not, it will cause an error. So there is actually a value for the compiler to evaluate, stored somewhere.
In the latter case, however, it is completely different. We are defining a variable named `x` for the first time, so there is no `x` to refer to, and hence no value to evaluate. The compiler expects there is no predefined `x` - even if there is, it will raise an error or a warning, not caring about what is stored inside.

Last edited 5 months ago by park

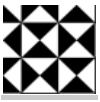
3 Reply

Vladimir

Reply to park [29] October 20, 2023 10:49 am

Thank you so much for your explanation!

0 Reply



Andy B

September 28, 2023 7:13 am

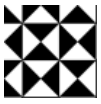
I'm having trouble with the distinction between full expressions and subexpressions.

From the lesson:
The subexpressions of `x = 4 + 5` are `x` and `4 + 5`. The subexpressions of `4 + 5` are `4` and `5`.
A full expression is an expression that is not a subexpression. `2`, `2 + 3`, and `x = 4 + 5` are all full expressions.

So the full statement is a full expression, that I can get.
I'm not sure I understand why `x` alone is not a full expression if the literals `4` and `5` are full expressions.
But then the subexpression `4 + 5` is also a full expression?

A little unrelated, but I recommend changing
"`2`, `2 + 3`, and `x = 4 + 5` are all full expressions."
to keep working with the "x = 4 + 5" example;
"`4`, `5`, `4 + 5`, and `x = 4 + 5` are all full expressions."

1 Reply



Andy B

Reply to Andy B [30] September 28, 2023 7:43 am

Sorry, I've immediately realized my mistakes.
The key things I missed were that subexpressions are used as operands.
I also didn't scroll up to see that "2" and "2 + 3" were their own separate lines in the example code block.

So on its own, `2 + 3` is a full expression, with subexpressions `2` and `3`.
If `x` was its own line, then it could be a full expression, but as part of `x = 4 + 5` it is a subexpression since it is an operand to `operator=` and the other subexpression `4 + 5`

Last edited 6 months ago by Andy B

0 Reply

Alex Author

Reply to Andy B [31] September 28, 2023 10:27 am

This. You got it.

1 Reply

ClickBrick

September 27, 2023 9:55 am

Why does

int z{ };
z = x;
std::cout << z - x <<

output 0 in question #3?

0 Reply

Alex Author

Reply to ClickBrick [32] September 28, 2023 9:34 am

`x` starts with value 6. When `z = x` executes, `z` is assigned the value `6`. Then we output `z - x`, which is `6 - 6`, which equals `0`.

0 Reply

Dominik Wolf

September 13, 2023 12:54 am

"Statement contains an expression (The right-hand side of the equals sign is an expression containing a single value, but the rest of the statement isn't an expression)."

I think that's not quite right. Not only the literal 5 is an expression. In this case "x = 5" would be an expression that evaluates to x and has the side effect of assigning (in this initializing) the literal 5 to the variable x.

Last edited 7 months ago by Dominik Wolf

0 Reply

Alex Author

Reply to Dominik Wolf [33] September 14, 2023 4:19 pm

This is a little tricky. `x = 5` by itself is an expression. It calls `operator=` with two operands: `x` and `5`.

`int x;` is a declaration statement. `int x = 5;` is a declaration statement with an initializer. This use of `=` is not a call to `operator=` -- it's part of the copy initialization syntax, not part of an expression. Only the initializer here (`5`) is an expression.

I'll add some of this explanation to the quiz answers for future readers.

0 Reply

Abdullah

August 26, 2023 1:52 pm

Can some one explain further, because I can't wrap my head around about what an expression is?

And is this considered an expression statement `int y {8};` or is it this `y {8};`

I'm confused??

Last edited 7 months ago by Abdullah

0 Reply

Alex Author

Reply to Abdullah [34] August 29, 2023 10:24 am

Expressions result in computation of a value.

`8` is an expression (a literal that evaluates to 8).

`y { 8 }` is not anything.

`int y { 8 };` is a definition for `y`, but it is not an expression statement because `int y { 8 }` is not an expression.

0 Reply

Abdullah

Reply to Alex [35] September 4, 2023 1:58 pm

Could you give me a code example of expression statements please.

0 Reply

Alex Author

Reply to Abdullah [36] September 5, 2023 3:40 pm

`y = 8;` is an expression statement because `y = 8` is an expression, and we've turned it into an expression statement via the semicolon.

1 Reply

Abdullah

Reply to Alex [37] September 6, 2023 12:10 pm

Thanks, but could this also count as an expression statement, or does the semicolon have to be written directly after the expression?

`num = { 6 }` or `num{ 6 }`

Last edited 7 months ago by Abdullah

0 Reply

Alex Author

Reply to Abdullah [38] September 7, 2023 12:40 pm

The semicolon turns the expression into an expression statement. An expression won't compile anywhere a statement is expected.

0 Reply

oldjak

Reply to Abdullah [34] August 27, 2023 11:57 am

An expression is any combination of variables, function calls, literals, operators, etc... that is calculated (evaluated) by the program to produce a single value.

Following that logic, I think only "y {8};" would count as an expression statement since "int y {8};" includes the "int" for declaring the variable's data type, making it more than just an expression inside that statement.

Would love to see what someone more experienced would tell us. Hard to find a good definitive answer to this on stack overflow too.

Last edited 7 months ago by oldjak

0 Reply

Swaminathan

Reply to oldjak [39] January 17, 2024 12:40 am

int y{8}; is a definition (also declarative statement). Here literal 8 is an expression that evaluates to 8. The other parts of the statement are not expression. You may ask why then y = 8; is an expression as it evaluates to y, but not y{8}. Because we instruct the program to not evaluate anything but place literal value 8 into the object y.

0 Reply

oldjak

August 23, 2023 5:10 am

Hello Alex,

```
1 Expressions involving operators with side effects are a little more tricky:
2
3 x = 5           // has side effect of assigning 5 to x, evaluates to x
4 x = 2 + 3       // has side effect of assigning 5 to x, evaluates to x
5 std::cout << x  // has side effect of printing x to console, evaluates to
  std::cout
```

Should the line 5 comment (regarding the expression std::cout << x) not say "has side effect of printing value stored in x to console" instead?

I understood what you meant, and I know that these were formatted to be short and concise. Just thinking it might confuse some learners.

Thank you for this great learning place. :)

Last edited 7 months ago by oldjak

0 Reply

Alex Author

Reply to oldjak [40] August 28, 2023 11:27 am

Done. Thanks for the suggestion.

1 Reply

Emeka Daniel

August 2, 2023 10:32 am

So, Expressions are a combination of C++ entities that evaluate to a single value.
While:
Expression statements are Expressions that are followed by a semicolon(they often return values that are discarded).
And:
Statements that contain Expressions are statements that are one part Expressions and one part not.

So the similarity between an Expression and a statement is that they both tell the computer what to do.
And the difference between them is what they tell the computer to do.

Reading this time and time again offers more insight.

2 Reply

Krishnakumar

Reply to Emeka Daniel [41] August 23, 2023 2:21 pm

Nice!

0 Reply

Emeka Daniel

Reply to Krishnakumar [42] August 24, 2023 2:22 am

Thanks.

0 Reply

Parker

July 5, 2023 10:10 am

It would be really helpful to have all these tutorials, not just this one, split into headings because I am blind and it is easier to navigate by headings with a screen reader.

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/
3. javascript:void(0)
4. https://www.learncpp.com/cpp-tutorial/developing-your-first-program/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/introduction-to-literals-and-operators/
7. https://www.learncpp.com/introduction-to-expressions/
8. https://www.learncpp.com/cpp-tutorial/keywords-and-naming-identifiers/
9. https://www.learncpp.com/cpp-tutorial/object-sizes-and-the-sizeof-operator/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-594010
12. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-593792
13. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-593245
14. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-593287
15. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-593180
16. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-593200
17. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-592853
18. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-592905
19. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-591318
20. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-591364
21. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-591396
22. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-590114
23. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-590116
24. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-589966
25. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-589967
26. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-589988
27. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-588964
28. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-588887
29. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-588906
30. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-587888
31. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-587889
32. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-587856
33. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-587102

34. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586191
35. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586357
36. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586561
37. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586625
38. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586694
39. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-586224
40. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-585939
41. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-584945
42. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/#comment-585965